Instructions on Using Git

1 Basics

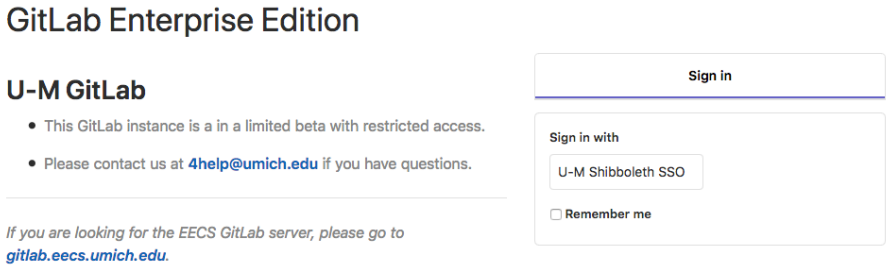
Git is currently the most popular version control system. It is a distributed system, as each checked out Git repository has full version history by default and it allows local commits. GitLab provides a user-friendly web interface to Git repositories. Before we even start, here are some ground rules for making (in Git terminology "push") changes in the repository:

- Do not push files larger than 1 Megabyte with the exception of SWMF_data and CRASH_data repositories that are designated to store large files (up to 100M). Gzip large ASCII files containing reference solutions or tables if they exceed 100k in size.
- Use "gitall status" to see a complete list of modified files. Make sure that the files are all committed and they are committed into the repository where they belong.
- Follow the coding standards. See http://herot.engin.umich.edu/~gtoth/SWMF/doc/SOFTWARE_STANDARD.pdf
- Do not break the code for others. The code should remain in working condition (your own application can be broken if it is only used by you). Make sure that the freshly checked out (cloned) code compiles. Run relevant tests with debugging options on and make sure they pass before push. Push all interdependent changes from all repositories. Do not push anything right before 7:00pm EDT/EST when the nightly tests start.
- If you pushed changes, check the nightly test page next morning around 10am at http://herot.engin.umich.edu/~gtoth/. Tests with changed results are CAPITALIZED. Check the log and see if your changes are responsible for breaking the tests. Fix the code or undo the changes. Do not leave broken tests for multiple days.
- Do not compromise the performance of the code or slow down the testing drastically. All functionality tests should finish within 5 minutes on a single core. Unit tests should be even faster.
- Document your source code. Document the changes in a way that it is meaningful and useful. Document the input parameters in the XML files. Create or modify nightly tests for new features and new applications. Keep those nightly tests working and up-to-date.
- If you break these rules often, your write privileges will be taken away. You will still be able to download (clone and pull), but not push. Your write privileges will only be reinstated if you can convince the maintainers that your Git usage practices improved.

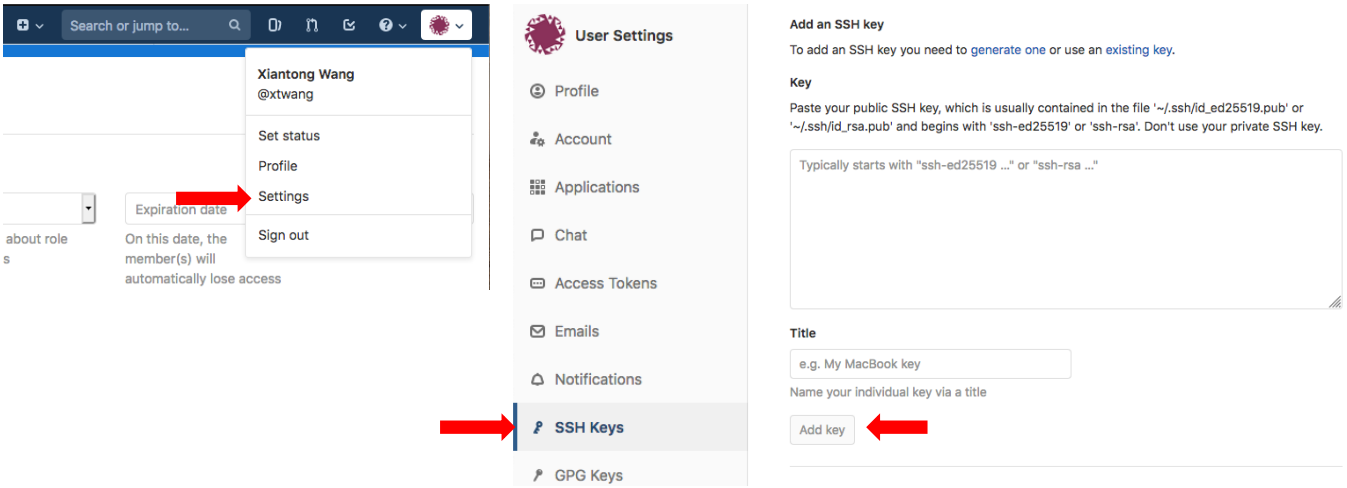
The University of Michigan provides GitLab running on a U-M server, which now hosts the SWMF software. Users will need a valid U-M uniqname to use this server. Please contact the SWMF developers if you need to get a U-M uniqname for this purpose.

1.1 Registration and initial setup

To create a GitLab account, log into gitlab.umich.edu with your U-M uniqname and password.



The next step is uploading your ssh key to GitLab and you will have to do this once from every computer where you want to access the GitLab repositories from. Go to your personal settings (top right corner) and choose SSH key on the left side of the webpage. Copy your ssh public key to the textbox and click “Add key”.



1.2 Use “gitclone” and “gitall” scripts

The following steps need to be done on all machines where you want to access the Gitlab repository from.

Copy or link the gitclone Perl script from share/Scripts/gitclone or download it from

https://gitlab.umich.edu/swmf_software/share/blob/master/Scripts/gitclone

into your executable path and make it executable with chmod +x gitclone. The script will clone a Git repository from the proper URL. Examples:

```
$ gitclone SWMF -history        # from gitlab.umich.edu:swmf_software
$ gitclone LATEX Papers/BIBTEX  # from gitlab.umich.edu:csem_software
```

```
$ gitclone GITM2 UA/GITM2       # from gitlab.umich.edu:swmf_software
$ gitclone GITM                 # from github.com/aaronjridley
```

See the help message (gitclone -h) for complete description.

Copy or link the gitall script from the cloned share/Scripts/gitall into your executable path to handle multiple Git repositories in subdirectories. Examples of use

```
$ gitall -v status
$ gitall pull
```

See the help message (gitall -h) for complete description.

Apply the following setting to make sure that you don't store outdated and potentially excessive (in terms of disk usage) Git history in the .git directory of the local repository:

$ git config --global pull.rebase true

This configuration setting will also be executed by the gitclone and gitall scripts.

The UM Gitlab access may be blocked by the university's firewall if there are many access attempts from the same IP address in rapid succession. If this happens, set the GITLABSLEEP environment variable to the number of seconds between successive accesses. For example, use

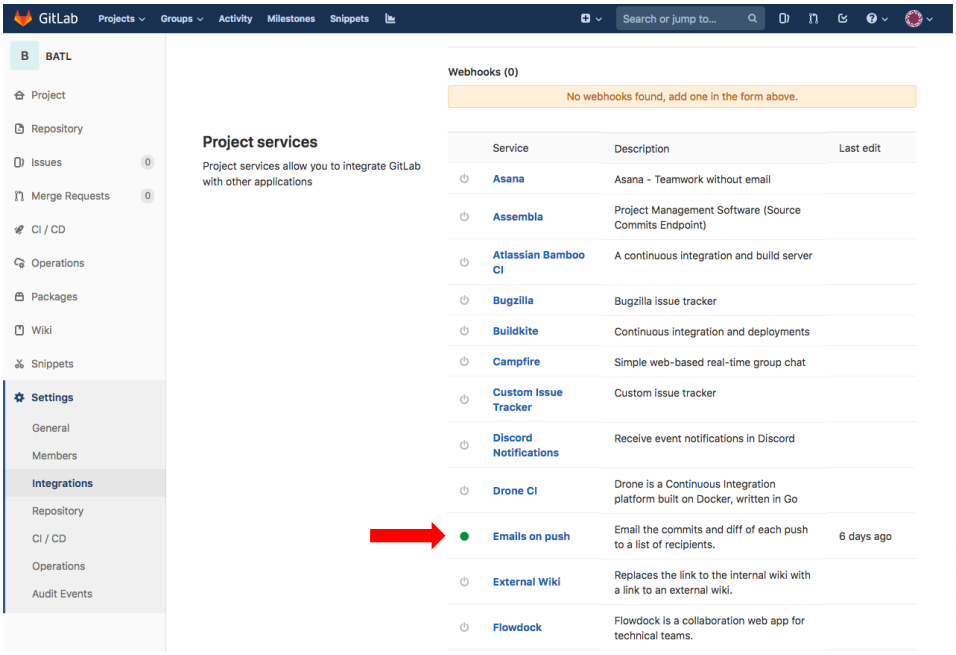
```
$ setenv GITLABSLEEP 5  # csh, tcsh
$ export GITLABSLEEP=5  # bash, ksh, zsh
```

in the appropriate shell initialization script (.cshrc, .bashrc, …). You need to wait a minute or two before the firewall resets after the blocking and you can start using gitall with the properly set GITLABSLEEP environment variable.

1.3 Add your email address to the notification list

If you want to get notified by email when there is a new commit to a GitLab repository, you can put your email address in the notification list. This needs to be done for each repository (called project in GitLab) individually.

Go to a project webpage, for example BATL, select "Settings -> Integrations" on the left. Then find "email on push" on the webpage (notice that you have to click on "active"):



And add your email address to the textbox.

1.4 Switch the current remote from herot to GitLab

If you have a working repository that uses herot.engin.umich.edu as remote and you want to preserve your current changes then you need to switch the remote to GitLab. If unsure about the remote repository, you can type

```
$ gitall -r
Git repo=herot:/GIT/FRAMEWORK/BATL  original 2020-03-07 c7654e1
```

which shows that the remote is herot. To change it to GitLab use

```
$ gitall -remote
$ gitall -r
Git repo=git@gitlab.umich.edu:swmf_software/BATL original 2020-03-09 bcce218
```

After this you may update and commit changes:

```
$ gitall pull                    # update from GitLab, merge if needed
$ git commit                     # commit your own changes
$ gitall push                    # now the change will be saved at GitLab
```

1.5 Simple test

When you finished all the setup, please try to clone the BATL repo from GitLab and see if you can successfully install it:

```
$ gitclone BATL
$ cd BATL
$ ./Config.pl -install
```

2 SWMF Git Repository

The version control information is stored at the top level of the working. To accommodate the SWMF structure where models can be checked out in stand-alone mode as well as part of the SWMF, we use multiple Git repositories. We created several Git repositories (SWMF.git, BATSRUS.git, GITM2.git, share.git, util.git and so on). All the SWMF related Git repositories are stored in the

```
gitlab.umich.edu:swmf_software/
```

directory. To make cloning (downloading) the SWMF and/or individual models, like BATS-R-US, easier, the *Config.pl* scripts can take care of cloning the Git repositories corresponding to the various subdirectories. In short, the SWMF git repository has multiple git repositories in its subdirectories.

3 Git Commands and Tools for Users

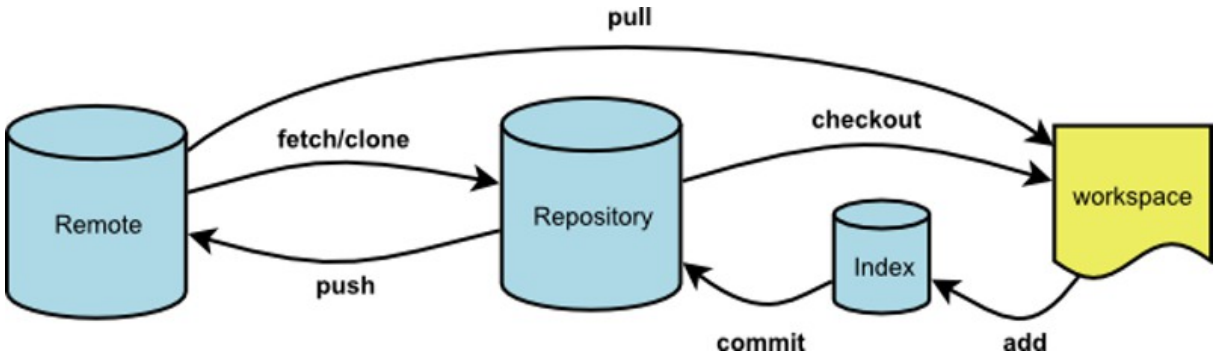


Figure 1 The working logic of Git system

3.1 Clone a remote git repository to a local one

$ gitclone REPONAME LOCALNAME

The optional LOCALNAME allows changing the name locally.

The command above clones the remote repository to the local machine with complete version history. However, the size of the local SWMF repository with all models included will be about 900 MB. Using --depth NUMBER allows limiting the version history, for example:

$ gitclone SWMF SWMF_NO_HISTORY --depth=1

clones the SWMF with the latest version only and the size is reduced to about 260 MB. This reduces disk usage and initial download time.

3.2 Tools developed for the SWMF with many git repositories

The improved *Config.pl* scripts now checks the existence of the necessary components, for example, the share/ and util/ subdirectories, and clones them from GitLab if necessary. In addition, the *Config.pl* script in the SWMF will also clone the SWMF models (like BATSRUS, AMPS, etc) during installation as needed.

We provide a script *gitall* in *share/Scripts.* This script will recursively search the git repositories and execute the git command passed to it. It is best to link or copy gitall into the execution path, so it can be used easily from any directory. **Notice that gitall only searches the subdirectories**. For example, to check the status of all the git repositories in the SWMF type the following at the top-level SWMF directory:

$ gitall status

By default, there is output only if there is a change in the repository. Using the -v flag (verbose)

$ gitall status -v

will show all the git repositories even if there are no changes in them. Other possible uses include

```
$ gitall fetch origin          # get latest version of the code from GitLab
$ gitall difftool origin       # compare local version with the version copied from GitLab
$ gitall merge origin          # merge local and copied versions
$ gitall pull                  # simply update the local source code from GitLab
$ gitall push                  # push all the changes to GitLab, remember to commit first!
```

Doing commit with gitall is not recommend since commit logs are typically independent in different repositories.

3.3 Checking the difference between the local git repository and the remote server

First of all, it is useful to define a visual difference tool, such as tkdiff, to be accessible by Git. Use the following command

$ git config --global diff.tool tkdiff

When other developers modified the project and pushed changes to the remote repository, it is useful to check the differences between the local and remote repositories. It can be done using gitall in top level of SWMF

```
$ gitall fetch origin                    # update the version information of the remote
$ gitall diff origin          # compare the local branch with 'origin' (the newest version in remote)
$ git difftool origin         # compare the local branch with origin using the difftool defined above
```

3.4 Update the git repository from the remote server

If the 'origin' branch was downloaded with 'git fetch origin', the changes can be merged with the local master branch using

```
$ gitall merge origin
```

Please refer to the internet for information about dealing with conflicts during the merging. If you are confident about the correctness of the remote repository, the fetch and merge steps can be replaced with a single command:

```
$ gitall pull
```

This will update your local repository (equivalent to fetch+merge). This is the most common and simple approach.

4 Working with the SWMF in Git

4.1 Working with the entire SWMF

Clone the core SWMF repository without models:

```
$ gitclone SWMF
```

Now you have a local SWMF repository without share, util and the physics models. The new features of Config.pl can take care of cloning (downloading) the missing pieces:

```
$ ./Config.pl                          # show SWMF info and clone util and share if missing
$ ./Config.pl -install                 # install SWMF with all models (clone the entire SWMF)
$ ./Config.pl -install=BATSRUS,PWOM    # install SWMF, clone GM/BATSRUS, PW/PWOM if missing
$ ./Config.pl -install=AMPS_PT         # reinstall SWMF, clone PT/AMPS if necessary
```

Note that models that are already present will not be cloned again during reinstallation.

4.2 Working with stand-alone models (BATSRUS, GITM2, PWOM …)

First clone the BATSRUS repository:

$ gitclone BATSRUS

Like in the SWMF, now you can use Config.pl to get all the necessary repositories, for example

$ ./Config.pl -install -compiler=gfortran

will automatically clone (check out) the share, util and srcBATL repositories into BATSRUS if not yet present. Note that these are independent Git repositories.

4.3 Using SWMF_data and CRASH_data

The large files are stored in the SWMF_data and CRASH_data repositories. These can be checked out into the home directory as

$ cd
$ gitclone SWMF_data
$ gitclone CRASH_data

This should be done before installing the SWMF or stand-alone model so that the data/ symbolic links can be properly created.

The information provided by now should be sufficient for a user who will not change the code.

5 Git Commands and Tools for Developers

5.1 Settings for line endings

If you're using Git to collaborate with others, ensure that Git is properly configured to handle line endings, on OSX use

$ git config --global core.autocrlf

5.2 Make changes to local repository

In essence use “git” followed by the Unix commands to remove and rename files and directories, use “add” to add a new file and “commit” to commit the changes into the local repository:

```
$ git rm -rf DIR1
$ git rm FILE1
$ git commit -m "removed FILE1 and DIR1 because …"
$ git mv FILE2 FILE3
$ git commit -m “renamed FILE2 to FILE3”
$ emacs FILE4
$ git status # make sure that FILE4 shows up as ‘Untracked’ so you are in the right repository!
$ git add FILE4
$ git commit -m “Created FILE4 that contains ….”
```

Without the -m flag the editor (defined by the $EDITOR environment variable) opens to allow logging the changes. Meaningful logs that describe the changes, the reasons for them and the consequences are extremely important and useful. You can also commit files separately and provide separate logs. This is recommended if the changes in different files are not related to each other.

The steps above affect the local git repository only. This allows storing multiple versions with a complete version history locally without making changes in the remote repository.

5.3 Undo changes to local repository

Sometimes an accidental or temporary change is made. There are various ways to undo the change depending on the status of the change:

(1) Change hasn’t been added to the index (no git add or git commit has been done)

```
$ git checkout FILE1          # undo changes in FILE1
$ git checkout .              # undo all changes in the repository
```

(2) Change has been added to the index but not committed (git add but no commit)

```
$ git reset HEAD FILE1      # use git reset to remove changed FILE1 from index
$ git checkout FILE1        # undo changes on FILE1

$ git reset HEAD            # use git reset to remove all changed files from index
$ git checkout .            # undo all changes
```

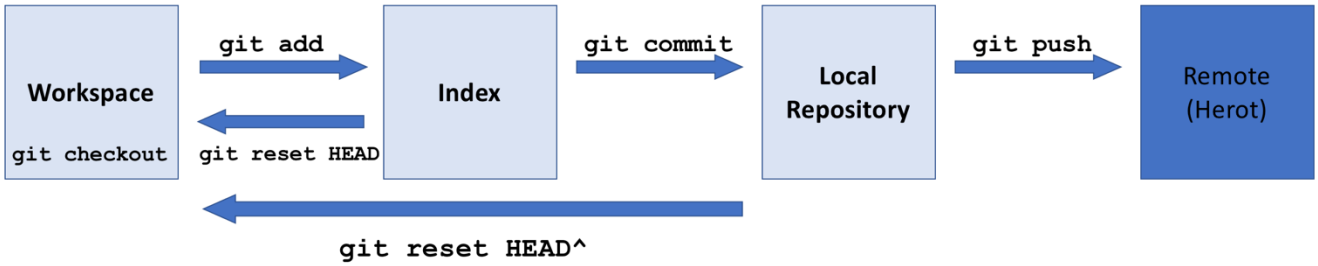
(3) Change has been committed but not pushed (git commit has been done)

```
$ git reset HEAD^           # revert the commit to current workspace
$ git checkout .            # undo all changes
$ git checkout FILE1        # undo changes on FILE1
```

(4) Change has been pushed (git push has been done)

Undo the modifications and commit and push them. Note that this really should not happen, because changes should be properly tested before being pushed to GitLab.

A figure to explain this:



5.4 Check the status of the present git repository

After working for a while, to check what you have done, in top level of the SWMF type

$ gitall status

Check the changes in all the modified files:

$ gitall diff # list all differences in the repository tree
$ git difftool # after examining a file with the difftool, git jumps to the next file

Note: gitall difftool is not recommended, as it will open multiple tkdiff windows.
Also, tkdiff can be used directly to examine the changes made to FILE1 (this only works in the top level directory of the Git repository):

$ tkdiff FILE1 # compare modified FILE1 to local master version
$ tkdiff -rorigin FILE1 # compare modified FILE1 to remote version fetched

5.5 Push changes to remote repository

After a period of local development, you may want to push the changes to the remote server Make sure you are on the master branch in the local repository as well as in the submodules.

$ gitall push # Note that gitall executes in the current directory and its subdirectories!

5.6 Get an old version of a git repository

Nightly tests of the SWMF can reveal bugs introduced during the code development. To identify the reason, it is necessary to compare versions of the SWMF at different dates. To get an old version, you need to clone the SWMF with full history. The default installation is without history

(to save download time and disk space). To get full history, use gitclone without the --depth parameter and install with the -history flag. The -date=DATE flag (which obtains full history and then sets the checkout date) can be used to get the code version of the required date:

```
$ gitclone SWMF SWMF_2018_06_19
$ cd SWMF_2018_06_19
$ ./Config.pl -clone -date="2018-06-19 19:00"
```

This will set the SWMF to the version tested at 19:00 EDT, June 19, 2018. Note that the resulting git repository will be in a status called “detached HEAD”. Changes made in this repository cannot be committed or pushed.

One can revert a single file to an earlier version to correct an incorrect commit, for example. Note the backticks in the example below:

```
$ git reset `git rev-list -1 --before="2018-06-19 19:00" master` FILE1
$ git commit -m “revert FILE1 to an older version because…” FILE1      # commit to the local index
$ git checkout FILE1              # use the older version to overwrite the version in the workspace
$ git push                        # push the reverted version to GitLab if it works correctly
```

Note that the version overwritten remains available in the Git history.